SOFTWIRED

Developing Publish/Subscribe Applications with iBus

- Technical White Paper

SoftWired AG
Technoparkstrasse 1
CH-8005 Zurich
Phone: +41-1-445-2370

info@softwired-inc.com, http://www.softwired-inc.com/

Exhibit B

1 What is iBus?

iBus is a pure Java™ *publish/subscribe software bus* allowing distributed components to exchange information via a variety of communication protocols and qualities of service. As a result, only a loose coupling of components is maintained, rendering components easier to reuse and applications easier to extend. In order to exchange information, components join one (or multiple) iBus *communication channels*. Components produce information and feed it into channels or receive information by subscribing to channels. Any kind of serializable Java object can be transmitted with iBus.

iBus supports reliable asynchronous *push* communication as well as synchronous *request/reply* style communication. The latter is analogous to CORBA and RMI and allows a component to explicitly request a piece of information via an iBus channel. Unlike CORBA and RMI, iBus request/reply style communication relies on multicast and is thus fault-tolerant. iBus employs TCP/IP and IP multicast to deliver information.

A distinguishing feature of iBus is its *Quality of Service (QoS) Framework*. It allows iBus to deliver information not only using the Internet protocol family, but also by protocols such as infrared, paging, and satellite communication. Thus iBus becomes interesting for applications running on client/server systems as well as for embedded systems. iBus supports tying a QoS to each iBus communication channel, i.e. reliable point-to-point communication, reliable multicast, best-effort communication, encryption, failure detection etc.

Below we provide an application example to demonstrate how a rather sophisticated financial data delivery application can effortlessly be implemented, using iBus. Finally, the iBus programming model and QoS framework is explained in more detail.

2 Example 1: StockQuote Producer and Consumer

The first example deals with receiving financial information from a source such as Reuters and its distribution via iBus to data visualization tools running on trader workstations. The data collector and the trader workstations are interconnected by a local area network allowing iBus to deliver information by its reliable IP multicast protocol.

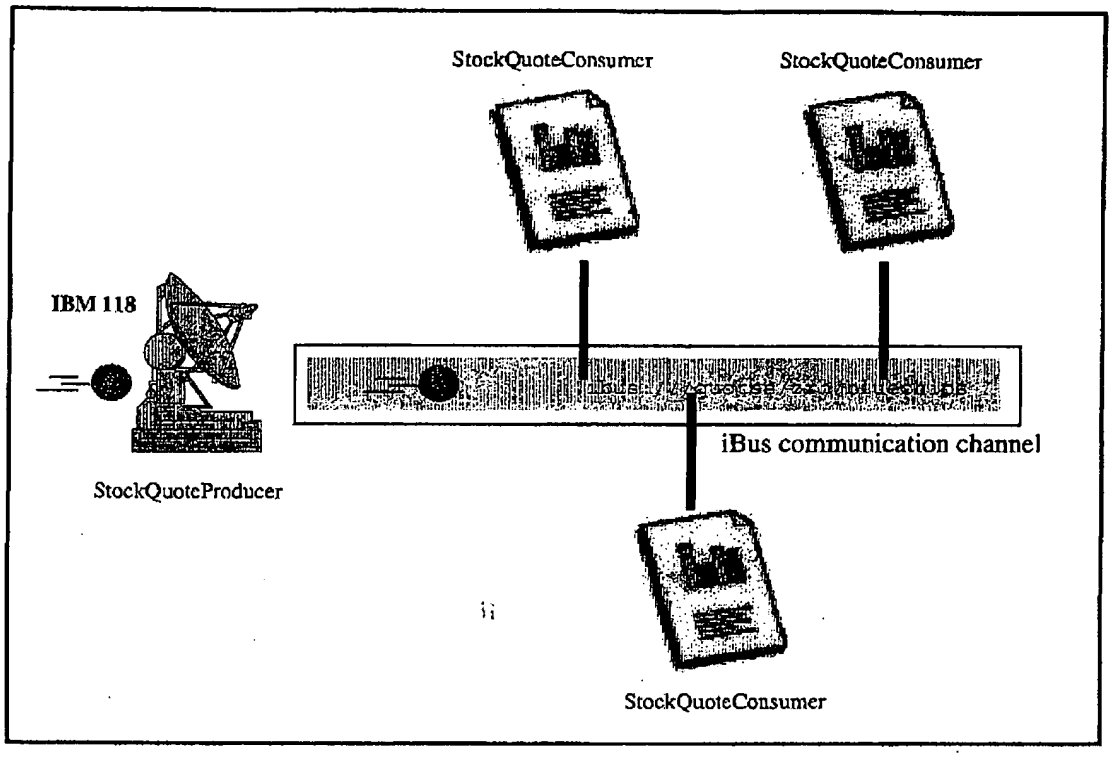

Figure 1: Stock quotes are encapsulated in Java objects and sent to a dynamic set of visualization tools.

In a typical trading floor environment many trader workstations plug into the network to receive financial information almost in real-time. Therefore, the distribution of information can not be accomplished by a point-to-point protocol such as TCP/IP without overloading the network. A multicast protocol avoids redundant network load and provides better scalability.

2.1 The `StockQuoteProducer` Application

The necessary code to implement a fully functional producer of financial stock quote events can be found below:

```
1 import ch.softwired.ibus.*;
2
3 public class StockQuoteProducer {
4 public static void main(String [] args) throws Exception {
5   int pushCounter = 0;
6
7   Publisher p = new Publisher();
8   Channel blueChipsZRH = new Channel();
9   ChannelURL url = new ChannelURL("ibus:///quotes/zrh/bluechips");
```

```
10  blueChipsZRH.setChannelURL(url);
11  p.setChannel(blueChipsZRH);
12
13  while(true) {
14    p.publish("Quote# " + ++pushCounter + " IBM 118");
15    System.out.println("Published quote# " + pushCounter);
16    try {
17      Thread.currentThread().sleep(1000);
18    } catch(InterruptedException e) {}
19  }
20 }
21 }
```

In order to push Java objects into an iBus channel an instance of the class `ch.softwired.ibus.Publisher` is created on line 7. Next, an instance of the `Channel` class is created (line 8) and an iBus `ChannelURL` is assigned to the channel (lines 9 and 10). Finally, on line 11 the `Publisher` object is tied to the channel. Now the `Publisher` is ready to transmit event objects on the channel.

As we will see in Section 4.1, a `ChannelURL` fully denotes an iBus channel by providing a hierarchical *topic* name for the transmitted information and an (optional) quality of service specification for the channel. In our case, the channel is used to transmit blue chips stock quotes from the Zurich stock exchange, hence the topic name `"/quotes/zrh/bluechips"`.

The infinite loop from line 13 to 19 is used to transmit fictitious IBM stock quotes via the channel. The quotes are encapsulated in `java.lang.String` objects, although any other serializable Java object can be transmitted as well. After publishing a quote a status message is written to the console and the publisher is suspended for one second.

2.2 The `StockQuoteConsumer` Application

An application tapping into the blue chips channel is written to receive stock quotes. The consumer application is structured like the producer application. Its main difference consists in creating a `PublishListener` object which is needed to receive the quotes transmitted on the channel. According to the JavaBeans model, iBus invokes the `PublishListener.handlePublishEvent` method when a quote is received on the blue chips channel.

```
1 import ch.softwired.ibus.*;
2 import ch.softwired.ibus.util.ThreadHelper;
3
4 public class StockQuoteConsumer implements PublishListener {
5 public static void main(String[] args) throws Exception {
6   StockQuoteConsumer consumer = new StockQuoteConsumer();
7   Subscriber s = new Subscriber();
8   Channel blueChipsZRH = new Channel();
9   ChannelURL url = new ChannelURL("ibus:///quotes/zrh/bluechips");
10  blueChipsZRH.setChannelURL(url);
11  s.setChannel(blueChipsZRH);
12
13  s.addPublishListener(consumer);
14
15  ThreadHelper.WAIT_UNTIL_EXIT.suspend();
16 }
17
```

```
18  public void handlePublishEvent(PublishEvent event) {
19    String quote = (String)event.getObject();
20    System.out.println("Got a stock quote: " + quote);
21  }
22 }
```

First, we need an instance of interface `PublishListener`. For simplicity we let class `StockQuoteProducer` implement the interface and we then create an instance on line 6. To comply with the JavaBeans model our `PublishListener` object needs to be connected to a local source of `PublishEvent` objects. To achieve this we create a `Subscriber` object on line 7 and tie it to the blue chips channel (line 8 to 11).

On line 13 we connect the listener with the subscriber. The peculiar line 15 is needed to suspend the `main` thread in such a manner that the consumer application will keep printing incoming stock quotes until interrupted by the user.

Lines 18 to 21 provide the implementation of the aforementioned `PublishListener.handlePublishEvent` method, called by iBus, to deliver a `PublishEvent` to the stock quote consumer application.

The following happens when the `StockQuoteProducer` enters the `Producer.publish` method:

1. The string object is serialized by iBus and is reliably transmitted on the blue chips channel by IP multicast.
2. The IP multicast datagram is received by the consumer's iBus middleware layers, deserialized, put in a `PublishEvent` object and passed on to the `Subscriber` object.
3. The `Subscriber` now passes the `PublishEvent` along to `StockQuoteConsumer.handlePublishEvent`.

2.3 Considerations

- **The system can be extended gracefully**: you can start as many quote producers and consumers in whatever order you like. It is not necessary to start the consumers before the producers.
- **Plug&Play of components:** producers and consumers can be relocated from one machine to another at run-time, without need to restart applications.
- **Light-weight channels**: for the sake of simplicity our application uses only one iBus channel. Applications can use many channels at the same time, and channels can be allocated temporarily or for the whole lifetime of the application.
- **Filtering**: a consumer receives only the events sent to the channels it is subscribed to. Hence producers and consumers need to agree on a `ChannelURL` in order to communicate.
- **Multiple protocols**: in our example, reliable IP multicast is used to deliver quotes to the consumers. IP multicast is the default communication protocol of iBus. This default can easily be changed, for example to TCP/IP, when the `ChannelURL` is created.
- **Scalability**: IP multicast implies that the same datagram is received by multiple destinations on the intranet thus eliminating the need to replicate the datagram once for every destination. The network load thus remains constant and better efficiency is achieved than with TCP/IP.
- **Fully distributed**: iBus requires neither naming services nor background daemons to administer its IP multicast channels. The iBus middleware is fully distributed and fault-tolerant.
- **Spontaneous networking** of components and devices is enabled in a fully fault-tolerant and scalable manner.

3 Example 2: SmartStockQuote Producer and Consumer

A disadvantage of the previous `StockQuoteProducer` application is that events are published even if no consumers are subscribed to the blue chips channel. Thus network bandwidth is wasted if no consumer applications are available. Fortunately, iBus offers a convenient tracking mechanism of channel subscriptions and unsubscriptions as well as application failures.

In the following example we illustrate a modified quote producer only publishing quotes if at least one consumer is subscribed to the blue chips channel and stopping publishing as soon as there are no more consumers. To achieve this we need to implement the iBus interface `ChannelViewChangeListener` and to register an instance of it with our blue chips channel. Now iBus will invoke the `ChannelViewChangeListener.handleChannelViewChangeEvent` method whenever a component (producer or consumer) plugs into the channel and whenever a component disconnects. The changes made to `StockQuoteProducer` are printed in boldface:

```
1 import ch.softwired.ibus.*;
2
3 public class SmartStockQuoteProducer
  implements ChannelViewChangeListener {
4
5 private static Object guard = new Object();
6 private static boolean listenerHere = false;
7
8 public static void main(String [] args) throws Exception {
9   int pushCounter = 0;
10
11  Publisher p = new Publisher();
12  Channel blueChipsZRH = new Channel();
13  ChannelURL url = new ChannelURL("ibus:///quotes/zrh/bluechips");
14  blueChipsZRH.setChannelURL(url);
15  p.setChannel(blueChipsZRH);
16
17  ChannelViewChangeListener l = new SmartStockQuoteProducer();
18  blueChipsZRH.addChannelViewChangeListener(l);
19
20  while(true) {
21    synchronized(guard) {
22      while (!listenerHere) {
23        System.out.println("No listener here, going to sleep...");
24        try {guard.wait();}
25        catch (InterruptedException e) {}
26      }
27    }
28    p.publish("Quote# " + ++pushCounter + " IBM 118");
29    System.out.println("Published quote# " + pushCounter);
30    try {
31      Thread.currentThread().sleep(1000);
32    } catch(InterruptedException e) {}
33  }
34 }
35
36 public void handleChannelViewChangeEvent
    (ChannelViewChangeEvent e) {
```

```
37
38  synchronized (guard) {
39    if (e.getNumListener() > 0) {
40      listenerHere = true;
41      guard.notifyAll();
42    } else {
43      listenerHere = false;
44    }
45  }
46 }
47 }
```

To keep the code minimal we let class `SmartQuoteProducer` implement interface `ChannelViewChangeListener` (line 3) instead of providing a separate class. We also need to provide the `handleChannelViewChangeEvent` method on lines 36 to 46. This method checks whether any consumers are running (line 39) and notifies the `main` thread (line 40 and 41). The variable `listenerHere` indicates whether at least one consumer is available on the channel.

The `main` thread checks the state of the `listenerHere` variable and is suspended on line 24 when there are no more consumers. If `handleChannelViewChangeEvent` detects the presence of consumers, it sets the `listenerHere` variable and resumes the `main` thread. The `main` thread then proceeds with the publishing of events. As soon as the last consumer has gone the `main` thread is suspended again, and so forth.

4 iBus Programming Model

The iBus API is event-oriented and JavaBeans compliant. In addition, synchronous request/reply style communication is provided. iBus thus offers the complete functionality of a publish/subscribe software bus plus the main features found in ORBs. The main API classes are:

- `Channel`: to represent a communication channel by which producers and consumers exchange events.
- `ChannelURL`: to name a channel such that producers and consumers can communicate with each other and assign qualities of service to channels.
- `Producer`: to inject objects into a `Channel` (push model)
- `Subscriber`: to receive events from a `Channel` and delegate those events to a local `PublishEventListener` object (push model).
- `Requester`: this class allows a client application to issue a blocking request/reply style operation.
- `Replyer`: this class allows a server to answer a request/reply style operation.

`Channel`, `Producer`, `Subscriber`, `Requester`, and `Replyer` are JavaBeans which can be deployed conveniently from within a visual development environment. The `ChannelURL` class is used to tie a QoS to a channel and will be examined next.

4.1 `ChannelURL` Class

iBus channel URLs obey the format

`"ibus:"[`*`QoS`*`":"]"//"[`*`address`*`][":"`*`address parameter`*`]"/"`*`topic`*

iBus URLs always start with the protocol descriptor `"ibus:"`. The next part is an optional *QoS* string indicating the transport protocol and reliability guarantees of the channel. If no QoS is given then reliable IP multicast is chosen by default.

Also, optionally, a channel *address* and *address parameter* can be specified. Thus an IP address, the wavelength of a satellite link etc. can be assigned to a channel. The format of the address and the address parameter depend on the QoS. The address parameter can be a port number, a baud rate etc. If no address is given then iBus will pick a default. In the case of IP multicast, a multicast address is assigned by computing the hash value of the topic string. This is accomplished without contacting a network-centric naming service.

The *topic* is a mandatory string describing the content of the channel. Some examples:

`ibus:///quotes/zrh/bluechips`: This URL only provides a topic, the other parts are left to their defaults. Reliable IP multicast is used. The IP address is computed out of the hash code of the topic.

`ibus://251.1.1.13/quotes/zrh/bluechips`: As in the previous example, but a specific IP address is assigned to the channel. This grants developers and system administrations full control over the allocation of IP multicast channels in their intranet.

`ibus://251.1.1.13:7771/quotes/zrh/bluechips`: A port number other than the iBus default (8733) is used.

`ibus:CRYPT:alias(reliable):///quotes/zrh/bluechips`: Data is encrypted and sent via IP multicast.

`ibus:alias(unreliable):///radiostations/RadioXYZ`: Unreliable multicast is chosen to transmit real-time audio in a best effort manner via UDP.

`ibus:TCP:///software_updates`: A TCP channel is used to push software updates via the Internet (server part).

`ibus:TCP://updates.softwired-inc.com/software_updates`: A TCP/IP channel is used to receive software updates from the Internet host `updates.softwired-inc.com` (client part).

`ibus://astra-satellite-X:89760000/quotes/zrh/bluechips`: Financial information is distributed via a satellite network.

5 iBus Quality of Service Framework

The capability to control qualities of services is one of iBus' distinguishing features. This is accomplished by embedding QoS strings into channel URLs. Altering the QoS of transmission channels thus only requires very minor modifications of the iBus application. For example, an application designed to deliver information via a satellite network can be developed and tested conveniently on PCs interconnected by a TCP/IP network. When the field tests start, the QoS of the channels are changed to reflect the satellite communication protocol. Hence the application can be "ported" from the IP network to the satellite network (and back again) very easily.

5.1 Protocol Stacks

The iBus QoS framework is open and can be extended by the developer with as yet unsupported qualities of service. However, most developers will simply deploy the existing iBus QoS and therefore do not need to be informed of what is described in this section.

Internally all communication is performed on behalf of so-called *protocol stacks*. A protocol stack represents one QoS and consists of a linear list of *protocol objects*. If an event object is published on a channel, it is passed to the protocol stack of the channel. The event object passes through the topmost protocol object in the stack and is then passed along to the next protocol object in the stack until it reaches the bottommost object. The bottommost protocol object delivers the event to the network interface. The event is then received by the bottommost protocol object of the consumer application and is passed up the stack. This process is depicted in Figure 2.

Regarding the iBus reliable IP multicast QoS a stack of five protocol objects is used. The topmost object (FRAG) fragments and reassembles objects that are larger than the underlying UDP datagram size (8 kb). FIFO performs first-in-first-out ordering of events. NAK detects when a datagram was lost on the network and requests its retransmission using iBus' unique *negative acknowledgements* reliable multicast protocol. REACH performs failure detection and tracks channel subscriptions and unsubscriptions. Finally, IPMCAST sends and receives UDP datagrams by IP multicast.

The protocol stack framework makes for higher modularity, better maintainability and extensibility of iBus and of iBus based applications. Since each protocol object performs only one specific task, protocol stacks can be composed that include only a subset of the fully reliable multicast protocol objects. Besides reliable IP multicast, iBus also provides protocol objects for TCP/IP, for data encryption, data compression, for persistent channels and more.

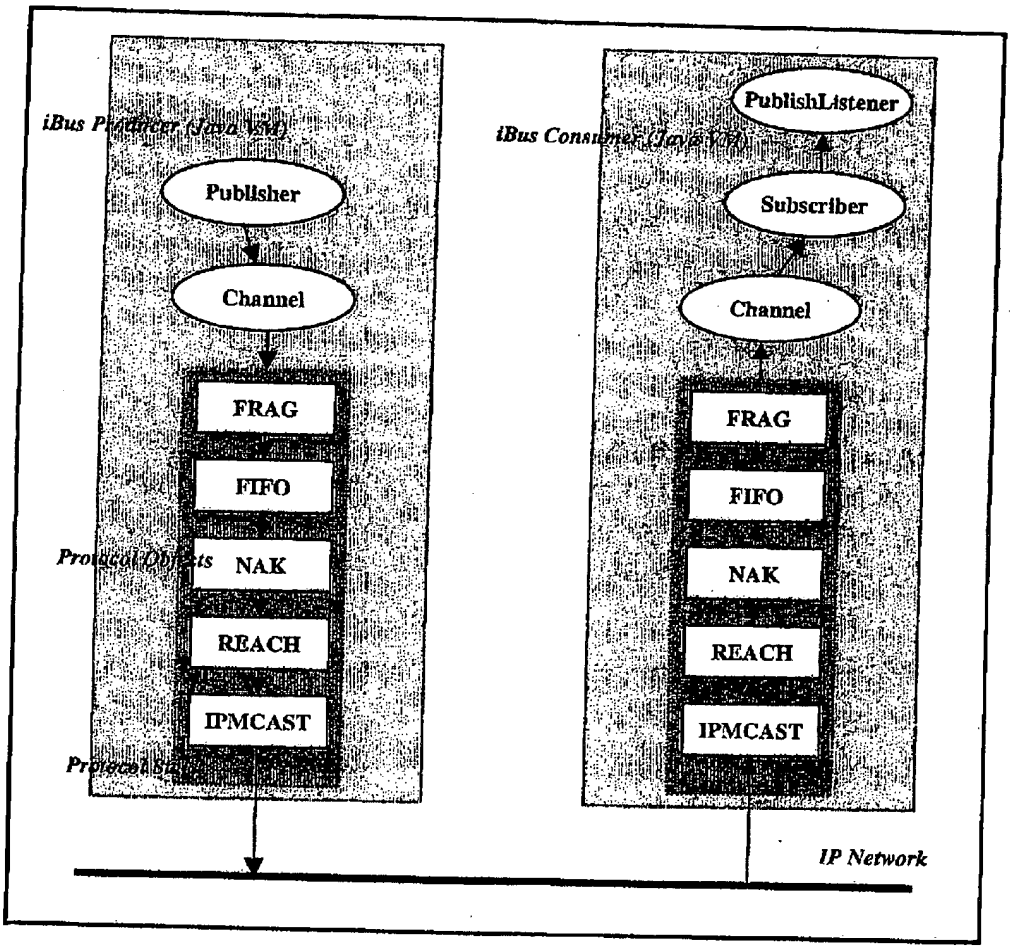

Figure 2: A producer and a consumer communicating by a reliable multicast protocol stack

5.2 Extending the QoS Framework

The iBus QoS framework can be extended by implementing new protocol objects. These objects can then be put on top of the existing iBus stacks, or completely new stacks can be composed altogether. For this the abstract base class `ProtocolObject` is provided. Protocol objects derive from this class. You can thus develop your own encryption protocol object, `MyCrypt`, by subclassing `ProtocolObject`. The protocol object can then be placed atop the iBus reliable multicast stack by specifying a channel URL like:

```
ibus:MyCrypt:alias(reliable):///topic
```

If such a channel is assigned to a publisher of events then the events will be encrypted by the `MyCrypt` object before they are sent via reliable IP multicast. At the consumer the events are decrypted before being passed to the application.

6 Conclusions

iBus is unique in that it realizes a fully featured publish/subscribe software bus and an extensible QoS framework purely in Java. iBus only requires the JDK and can be installed and deployed very easily.

The API is simple yet powerful, allowing complex distributed applications to be developed and extended quickly. iBus' fully distributed architecture and its small footprint make it attractive for both desktop and embedded systems.

7 Obtaining the iBus Software

Trial versions of iBus can be downloaded from the iBus home page at
`http://www.softwired-inc.com/ibus`

Please contact us if you need more information:
`mailto:info@softwired-inc.com`